

MISOSYS Disassembler - Disk version III

DDDDDD	SSSSSS	MM	MM	BBBBBB	LL	RRRRR
DD DD	SS SS	MMM	MMM	BB BB	LL	RR RR
DD DD	SS	MMMMMM		BB BB	LL	RR RR
DD DD	SSSSSS	MM	M MM	BBBBBB	LL	RRRRR
DD DD	SS	MM	MM	BB BB	LL	RR RR
DD DD	SS SS	MM	MM	BB BB	LLLLLL	RR RR
DDDDDD	SSSSSS	MM	MM	BBBBBB	LLLLLL	RR RR

Copyright (c) 1983 by Roy Soltoff

Programmers have probably been disassembling machine code programs since the time programs were being "hand" assembled. What is a disassembly? Simply the reverse process of assembly - taking a program or a piece of a program and translating it back to an assembly language state. This disassembler is a tool for helping you with the process. DSMBLR III is a third generation product. This tool provides extensive capabilities such as direct disassembly from CMD disk files, automatic partitioning of output disk files, data screening for non-code regions, and full label generation. DSMBLR III even generates the ORGs and END statement - the complete ball of wax.

Surely a tool of this capability should be difficult to use. Not so! You will find that the use of this disassembler - even by a beginning assembly language programmer - will be paying handsome rewards with the ease of its use and clarity of these instructions. DSMBLR III is a professional tool for your use.

TABLE OF CONTENTS

GENERAL	2
DISTRIBUTION DISKETTE	2
EXECUTION INSTRUCTIONS	3
CONTROL FUNCTION OPTIONS	4
INPUT MODE SPECIFICATIONS	6
DISASSEMBLY ADDRESS PROMPTS	7
SCREENING DATA ENTRY	8
OUTPUT COMMAND OPTIONS	11
DEVELOPING A SOURCE PROGRAM	15
DEVELOPING SCREENING DATA	16
APPENDIX	19

GENERAL

=====

The MISOSYS disassembler is a machine language labeling disassembler that supports direct disassembly from "CMD" files. The generated output can be directed to a line printer, the video screen, or automatically partitioned into multiple disk files. DSMBLR-III functions with the Radio Shack TRS-80 Model I or Model III microcomputers. PRO-DUCE functions under LDOS 6.x.x. This disassembler operates in two passes in order to incorporate symbolic labels in the source output. The symbolic labels are generated for address and 16-bit numeric references with the start-to-end user disassembly request or the scope of the CMD file. All address references not coincident with the start of an instruction's address within the range of the disassembly are output as equates (EQU) which can be optionally suppressed.

You are assumed to be familiar with Z-80 assembler mnemonics as specified in the ZILOG, "Z-80 ASSEMBLY LANGUAGE PROGRAM MANUAL", 3.0 D.S., REL 2.1, FEB 1977. Many texts can be located which provide various insights into Z-80 assembly language programming. Do not overlook articles on assembler routines appearing in the magazine and journal media.

This version provides a disk file output in standard un-numbered ASCII format compatible with the MISOSYS editor/assembler EDAS. Options are provided to add a file "header", line numbers, or a colon (:) after labels except those defined with EQU. These options permit the output file to be altered to suit other assemblers.

DISTRIBUTION DISKETTE

=====

The Disassembler Version III is a machine language program supplied on a data disk. For Model I/III users, DSMBLR-III is supplied on a 35-track single-density LDOS data diskette [Model III TRSDOS users must use the TRSDOS "CONVERT" utility to transfer files from the distribution diskette to their SYSTEM diskette. Model 1 TRSDOS 2.3 users MUST READ THE APPENDIX]. The LDOS 6.x.x compatible PRO-DUCE is distributed on a 40-track single density diskette. Each distribution diskette contains two copies of the disassembler. One is named "DSMBLR/CMD" with the other is "DSMBLER/BKU" and is a backup copy of the former. Three other files are included. "SCRIPSIT/TXT" is a text file containing screening data for the Radio Shack Model I version of SCRIPSIT/LC. By disassembling a copy of SCRIPSIT/LC using this text file for screening data, a good set of source files are output.

Two other files are "BINHEX/CMD" and "BINHEX/TXT". These files are supplied to serve as an example of program disassembly. The section on "Screening Data Entry" explains the use of these files.

LDOS is a trademark of Logical Systems, Inc.

TRSDOS is a trademark of Tandy Corp.

EXECUTION INSTRUCTIONS

=====

The disassembler is executed by issuing the command:

=====

```
DSMBLR [filespec] [(colon,header,lines=xx,number,size=yy)]
```

where:

- filespec** - Is the filespec of the disk file you want to disassemble. If you omit the file extension, "/CMD" will be assumed. The filespec is optional. Program prompts will query you as to the input mode if the filespec is omitted.
- colon** - Specifies that you want the disassembler to append a colon (:) to symbols in the label column that are not defining EQUates.
- header** - Is a parameter to force any output file to generate a header string prior to the first source line. The header will consist of X'D3' followed by the first six characters of the output filespec.
- lines=xx** - Specifies that you want the line printer output to issue a form feed (X'0C') after "xx" number of lines have been printed. The disassembler maintains an internal line counter that increments on each carriage return. The default setting of "lines" is 56. Thus, if you set your form paper to start printing on the fifth line from the top of the form, the disassembler will maintain a five-line margin top and bottom on 11" paper.
- number** - Specifies that you want the output disk file to add line numbers on each line. The line numbering starts with 00001 and increments by 00001 for each line. The format used is a 5-character ASCII number with the high-order bit set. The number is immediately followed with a space. Note: video and printer output is always numbered.
- size** - Specifies the maximum size of the output disk file before partitioning. Size will default to 12K. See the text on output file partitioning.

Parameters may be abbreviated to their first letter.
Command line entries in square brackets "[]" are optional.

CONTROL FUNCTION OPTIONS

=====

The disassembler supports options to control the disassembly. These options are entered in response to the following message:

```
Control function: <D>OS exit, <C>lear table, toggle <E>quates  
<S>ystem tape, <T>est tape (D,C,E,S,T)? >
```

The <S> and <T> options are only available on the Model I and III computers. Each time you enter an option, the prompting message will re-appear (unless you have selected <D> - exit to DOS). When you have completed your option specification, depress <ENTER> and the disassembler will proceed to the next prompt. A discussion of the purpose of each option follows.

Each time you complete a disassembly, the disassembler will return to the control function prompt. The <D> option is used to return control to the DOS command interpreter. This is the way in which you exit the disassembler. The "DOS READY" message will be displayed.

DSMBLR is a two-pass disassembler. The first pass disassembles the target program and builds a table used for the generation of symbolic labels. Output is not generated during this pass. After the first pass completes, the symbol table is not reset until cleared. Subsequent disassemblies proceed immediately with pass two - the pass which produces output. The first pass is performed only when the symbol table buffer region is "cleared". By issuing the <C> option the buffer is cleared and the following message is displayed:

```
Symbol table cleared
```

When the disassembler first executes, it is placed in a mode that simulates an un-cleared symbol table. The SYMBOL table is regenerated only when the table is cleared using the control function <C>. This is done to enable you to quickly scan a memory region without having to wait for the first pass to complete. If you choose to disassemble directly from disk, you do not need to specify the <C> option as it is automatically performed.

It is common practice to define program constants and address references to other programs at the front end of a source program by means of equate statements (with the assembler pseudo-op, EQU). When the target program contains address references that precede the start-of-disassembly, these references will be output as EQU statements. Equates are also generated for each symbolic label found in the symbol table that does not correspond to the start of an instruction. These labels are output in the form:

```
LABEL EQU $-n
```

where "n" is the offset to the label address from the current program counter. Equates are also generated for all address references that extend beyond the end of the target program.

You may chose to suppress the generation of equates in the disassembler's output by using this option. Equate generation will be either on or off. A flag control is used to indicate the ON or OFF mode. You reverse the flag's status every time you enter the <E> option. As is the case with the <C> option, a disassembly directly from disk will force the flag to be ON.

The status of equate generation is shown each time the Control Function prompt is issued.

The <S> option is available only on the Model I or Model III computer. It will load a SYSTEM program into memory. It is strongly suggested that you transfer a SYSTEM tape program to a CMD disk file and directly disassemble the disk file. The <S> option will identify the program's FILENAME, its STARTing address, its ENDing address and it's TRANSFER address (the location that control will be transferred to after loading a SYSTEM program via the SYSTEM command and issuing the "/" <ENTER>). The program's FILENAME and address information will be displayed in the message:

FILNAM: START=xxxx, END=yyyy, TRANSFER=zzzz

where xxxx, yyyy and zzzz are displayed in hexadecimal. Also, if the program loads without a checksum error, the START and END variables will be retained for automatic use in the disassembly. Caution should be observed if you suspect that the program may overlay the operating system. If the program loads below 5200H, your system will probably crash. You will be informed if the program will overlay the disassembler. It is suggested that when in doubt, use the <T> option first.

The <T>est option operates just like the <S> control function. However, since you may want to discover the address load information without physically loading the program, this command will do just that. The information is identified, but the program is not loaded into memory. The START and END variables are updated. Remember, it is best to transfer a tape program to disk and use disassembly directly from disk.

INPUT MODE SPECIFICATION

=====

Whenever you exit the Control Function prompts, you will be prompted to enter the mode of input. Two modes are available, disk (D) or memory (M). The prompt message is as follows:

Input: <D>isk or <M>emory (D,M)? >

If you are disassembling from memory, you will receive prompts to specify the starting, ending and relocation addresses to be used for the disassembly. These prompts are shown in the next section entitled "Disassembly Address Prompts". If your disassembly is from a CMD-type load module file (by entering <D>), you will receive a prompt to enter the filespec with:

Enter filespec? >

If you do not specify a file extension, "/CMD" will be used. If the file you specified is opened properly, the disassembler will interrogate you for screening data. This is discussed in the section entitled "Screening Data Entry".

DISASSEMBLY ADDRESS PROMPTS

Whenever you request a disassembly from memory (disassemblies from disk will automatically use the entire CMD file), you will be prompted to enter the storage locations of the program you want to disassemble. The addresses are entered in hexadecimal. Full line input control keys (backspace, line delete, etc.) are supported as in BASIC or DOS command input. In addition, you may enter the value without leading zeroes (0000 as 0, 06CC as 6CC or 6cc, etc.). The hexadecimal numbers X'A' through X'F' may be entered in lower case as well as upper case. These prompts appear as follows:

Start address = >

Enter the memory address at which the disassembly should begin. This will be the first memory location that will be disassembled. If the <S> control function was used to load a SYSTEM program, this value would be automatically set to the programs START address.

End address = >

This is the memory address at which disassembly should cease (Note that disassembly will run from START up to but not including END so END should be one memory position beyond where you want to stop the disassembly). For example, if you want to disassemble memory from X'0000' through X'2FFF', END should be entered as "3000". The disassembler will not function properly if END is entered as "FFFF". Similar to START, this variable will be set to one greater than the end address if the program to disassemble was loaded with the "S" control command.

Reloc address

If you had to move the program that you are disassembling (termed the target program) to an address area different from where it originally loaded because it would have overlayed (loaded into the same region as) the disassembler, the START address should be entered here. For example, if the target program originally loaded from 5000H through 5500H and you move it to load at 7000H through 7500H, then use START=7000, END=7500, RELOC=5000. This feature is useful to recover proper address references to code that may have been relocated to a higher or lower address in order to eliminate conflict with the load point of the disassembler. Programs may be moved by using CMDFILE, the extended DEBUG, or other such utility.

Address entries are retained by the program until changed by entering new values. Therefore, subsequent disassemblies using previously entered address information can be performed just by depressing the <ENTER> key. Also, responses to three address prompts may be entered on one line by separating each with a comma. For example:

Start address = >150,400,150

will input STARTing address of X'150', ENDing address of X'400', and RELOCation address of X'150' (a relocation address of X'150' would also be used by omitting the relocation entry and depressing <ENTER> in response to the "Reloc address" query).

SCREENING DATA ENTRY

=====

Just about every program that you will disassemble has segments that are actual code and other segments that are data. The disassembler will assume that all segments are actual code unless told otherwise. The "screening data" entry is the way in which you tell the disassembler what segments of the program are to be interpreted as data regions. The disassembler will prompt you with the message:

Enter screening filespec or <ENTER>

If you have NOT prepared screening text, depress <ENTER> and the disassembler will proceed to the output command prompt. If you have prepared a screening text file, enter its file specification. If you omit the file extension, "/TXT" will be assumed. If you are using LDOS or another LDOS compatible DOS, you may enter "*KI" as the specification and enter your data from the keyboard. Your entries will be echoed to the video screen; however, all characters are treated as text data - backspace is non-functional. Using *KI is a quick-and-dirty method of entering a very brief amount of screening text.

You must arrive at the addresses of the "segments" by an analysis of the target program. For instance, a first disassembly to the video screen will easily identify literals since the ASCII equivalent of the object code is displayed. Make note of the address ranges on a piece of scrap paper to be used in building a text file of screening data fields. The "smarter" you are in assembly language, the easier it will be to identify word and byte data. The section entitled, "Developing Screening Data" will provide hints and techniques to aid you. Once you build your text file, repeat the disassembly process to "purify" the resultant output. With a little bit of effort, you can rapidly construct a perfect source code image.

The address ranges are entered in the following formats:

aaaa-bbbb	=	Treat as data, all bytes in the range X'aaaa' through X'bbbb' inclusive (aaaa <= bbbb).
-cccc	=	Treat as data, all bytes in the range X'0000' through X'cccc' inclusive.
dddd-	=	Treat as data, all bytes in the range X'dddd' through X'FFFE' inclusive.
eeee	=	Treat as data, the byte at address X'eeee'

If during a disassembly, a "properly" decoded instruction extends into a screening data range entered, data will be interpreted starting with the next address.

Since each program's data structures are unique, the disassembler accepts screening data entries in a loose format. The fields are entered as plain text using the EDAS editor, most word processors that provide ASCII output, or even via the BASIC program, TEXT/BAS, listed in the APPENDIX. Each field is separated by a comma delimiter. Carriage returns may be entered and take the place of a comma (that means that if you use discrete lines, the entire field must be contained on one line. Any entered spaces are ignored.

The input stream is terminated by entering a period (.) after the last field entry.

Data usually take one of three forms: literal fields commonly called strings (words that you can read - i.e. messages, prompts, etc.); byte fields of varying length with each byte a distinct value (tables, conversion codes, one-byte length specifiers), "confusion" bytes (hex values placed to alter the sense of following code depending on entry point; and "words" of varying length (16-bit values commonly used as pointers, arithmetic values, etc.). The disassembler recognizes certain prefix specifiers to force the data generation to literal, byte, or word formats.

Prefixing the screening field with a dollar sign (\$) will force literal decomposition with the output formed into "DB 'string'" (equivalent to DEF.M) pseudo-OP declarations. A pound sign (#) will force the data to be decomposed into words using "DW Mxxxx" (equivalent to DEF.W) pseudo-OP declarations. The DW operand fields will be in symbolic name ("label") format - with the resulting 16-bit values forcing an entry into the symbol table. The default will be decomposition into "DB xxH" declarations if no prefix is specified.

Where a literal has been forced, the disassembler maintains a range check for the characters. Valid literal characters are in the range X'20' - X'26' and X'28' - X'7E'. If a character value is outside this range, the decomposition will automatically revert to "xxH" format starting with that character and continuing until a character within the literal range is detected. For example, The byte sequence:

```
4C 61 6E 27 74 0D
```

will decompose to the pseudo-OP declaration:

```
DB<tab>'Can',27H,'t',0DH
```

Note that multiple operands are generated on a single line with each offset by a comma. The MISOSYS assembler accepts this syntax on DB and DW pseudo-OPs. It permits intermixing byte and literal operands with the DB pseudo-OP. If the assembler you will use to assemble the output does not support this construct, you will have to separate the line into multiple lines once you obtain your output file.

The disassembler will output approximately 18 characters in the operand field of a line. If the screening range is such that the decomposition would exceed that limit, a subsequent line is generated. Also, any labels that would be addressed in the scope of the DB or DW field will be output immediately following the line. This would appear as "LA8EL EQU \$-n", where "n" is the offset from the current program counter. If the label is valid, you may want to split your screening field into two fields starting the second at the label address and ending the first one address before the label.

A sample screening data input is as follows:

```
5228-5229,$5384-53aa,#5829-5832,5416
$5b20-5b3d,5f67-5f68.
```

Note that a line terminated by a carriage return does not have a terminating comma. Also, the final character is a period (which may be followed by a carriage return). Note also that the fields do not have to be in ascending order (however, a range specification must be LOW-HIGH, i.e. the first address entry must be less than or equal to the second address entry. The disassembler will sort the fields by address after they are parsed. This makes it easy to add to the screening data as you "purify" your disassembly of a particular program.

The distribution disk contains a few screening text files. One is for Radio Shack Model I SCRIPSIT (SCRIPSIT/TXT). Another is BINHEX/TXT which is the screening data for the sample public-domain program, BINHEX/CMD, which is included on your disassembler diskette. BINHEX is a Model I/Ill program that converts a binary "CMD" file to ASCII in a HEX format (similar to Intel's HEX format). BINHEX was originally written in BASIC by Timothy Mann to use for transmitting CMD files over a 7-bit communication's line. It was further modified and then compiled using Bill Stockwell's BASIC/S compiler. BINHEX/CMD is in the public domain and is included for you to disassemble - as an exercise. The BINHEX/TXT file has been developed by MISOSYS to be used as screening data while disassembling BINHEX/CMD. If you need the experience, it is suggested that you "play" with the disassembly of BINHEX prior to using the supplied text file as screening data. Develop your own screening data file and compare it to ours.

The SCRIPSIT/TXT file can be used by those folks with access to a Radio Shack Model I version of SCRIPSIT/LC. When used as a screening data file, it should produce a useful set of source files - uncommented!

OUTPUT COMMAND OPTIONS

=====

The output device to receive the disassembly output is determined in response to the prompt:

```
Output: <R>eview, <S>creen, <P>rinter,  
<T>ape, <D>isk (R,S,P,T,D)? >
```

The <T>ape option is available only on the Model I and Model III computers. Select one of the devices by entering its respective letter.

During the disassembly, the byte value of instructions that have a byte-value operand will be displayed in either of two formats depending on the value of the byte. Bytes in the range X'20'-X'26' or X'28'-X'7E' will be output as literals in the form, "'c'". All other values are displayed in the form, "xxH", or if the value is in the range X'A0'-X'FF', "0xxH". The byte values of index instruction offsets are output in the non-literal format only. Also, the port number of IN and OUT instructions is kept non-literal. In most cases, this display format provides more meaningful information than displays strictly in the non-literal format. Sometimes, the literal presentation looks foolish. Since the display format is chosen by ranging, please accept the compromise approach chosen for the disassembler. You will notice output displays such as:

100C 2814	02442	JR	Z,M1022	(.
100E FE45	02443	CP	'E'	.E
1010 2810	02444	JR	Z,M1022	(.
1012 FE44	02445	CP	'D'	.D
1014 280C	02446	JR	Z,M1022	(.
1016 FE30	02447	CP	'0'	.0
1018 28F0	0244d	JR	Z,M100A	(p
101A FE2C	02449	CP	'.'	.,
101C 28EC	02450	JR	Z,M100A	(1
101E FE2E	02451	CP	'.'	..
1020 2003	02452	JR	NZ,M1025	.

which provide a greater ease in understanding the logic of a program.

<R>EVIEW

The <R>eview function will produce a screen listing identical to that discussed under <S>creen. The exception is that the listing will be displayed in a continuous scroll instead of a screen at a time. The scrolling may be temporarily suppressed by depressing the <PAUSE> (or <SHIFT-@>) key. Any character entry except <PAUSE> will resume the scrolling.

<S>CREEN

The <S>creen output is directed to the video CRT. Output is scrolled for 24 lines (16 on Model I or Model III), then paused. The next "page" commences upon depression of any keyboard key. Depressing <BREAK> will interrupt output and return you to the prompting message.

The output display consists of the following references:

1. Effective memory address of the instruction.
2. Contents of memory starting from the instruction's physical memory location for as many bytes as the instruction's length. Output is in hexadecimal.
3. Sequential line number, in decimal, starting from 00001 and incremented by 00001.
4. A SYMBOLIC LABEL, where referenced as a 16-bit or relative value by the program to be disassembled, consisting of the address referenced preceded by the letter "M".
5. The disassembled instruction using ZILOG mnemonics. The tab character between the OP code and the OPERAND is expanded for screen display.
6. Character output (in ASCII) of the instruction's hexadecimal values. Bit 7 is stripped from each byte prior to display in order to better identify character strings that utilize bit 7 for "begin-string" or "end-string" detection. Non-printable characters are converted to a period.

<P>RINTER

This function will provide the same output as the <S> function except that the output is directed to the LINE PRINTER. The output is printed 56 lines per page or other amount depending on your optional LINES parameter. Each page is numbered sequentially starting from 00001 and incremented by 00001. A heading which labels each column is provided on each page.

When the printer command is entered, the program will request you to enter a title and position the printer paper to receive the output listing. The prompt:

Ready printer and enter title

will be displayed. If you are using the 56 lines per page default, your paper should be positioned to start printing on the sixth line of the page. This will provide a top and bottom margin of five lines each on eleven inch paper which takes 66 lines per page. You may enter a title of up to twelve (12) characters which will be placed in the heading on each page of printed output. After depressing <ENTER> following the title, the disassembly will automatically start. By depressing the <BREAK> key at any time during the printing, the output will be interrupted and you will return to the prompt message (Model I or Model III users can only interrupt the printing while the printer is in a "ready" state).

<T>APE

This command, available only on the Model I or Model III, will create a source cassette tape suitable for loading into the Radio Shack cassette Editor Assembler or Microsoft's EDTASM+. After entering the <T>ape command, you will be prompted to prepare the cassette for writing with the message:

Ready cassette and Enter (Model I)
Ready cassette and Enter <H,L> (Model III)

Depression of the <ENTER> key will cause the disassembly to start. If you are using a Model III machine, you must select the appropriate speed of the tape file by entering either "H" for 1500 baud or "L" for 500 baud generation. You must specify the parameters HEADER and NUMBER when executing the disassembler to properly construct an output tape. The output consists of:

1. The 5-digit ASCII line number,
2. The SYMBOLIC LABEL (or tab if a label is not required),
3. The disassembled instruction. The tab character between the OP code and the OPERAND is not expanded.

The tape is created in blocks consisting of 256 lines of output per block. File names are assigned sequentially. The first is "BLOCKA", the second is "BLOCKB", etc., incrementing the sixth character by one letter for each block. A five (5) second blank segment is written between each block to provide a manual search capability. An asterisk (*) blinks in the upper right hand corner of the screen (3C3FH) for every two lines of output. The starting address of the block will be output to the screen. Depressing the <BREAK> key will interrupt the tape output only during the period of asterisk blinking.

<D>ISK

The <D> function provides the capability of generating a source disk file which can be loaded into the MISOSYS editor assembler, EDAS. The file(s) will be un-numbered and un-headered unless the NUMBER and/or HEADER parameters were entered on the disassembler execution command line. You will be prompted to enter the filespec with the prompt:

Enter filespec? >

If you omit the file extension, "/ASM" will be used as a default. After entering the desired filespec, the source file will be created. It is entirely possible that a disassembly could create a file larger than will load into EDAS depending on your system's memory configuration and the memory region or disk file being disassembled. In order to make sure that this does not happen, the disassembler will automatically partition the output into multiple files once a file reaches the maximum size as specified with the SIZE parameter.

In order to accomplish the output partitioning, the disassembler first makes sure that the diskette receiving the output has free space sufficient to store the maximum size file. As each line is written to the output file, the disassembler checks to see if the last sector of the file has been reached. Processing continues if the output has not reached the last sector; however, if the last sector has been reached, the EOF character (X'1A') is written and the file is closed. If the output filespec originally contained not more than seven characters, a new filename will be automatically constructed by appending 'A' to the filename. Assuming all goes well, you will be informed of the automatic generation of this file by the message:

Output file is full. Creating file: filespec

The new file will be created and processing will continue until this new file reaches the maximum SIZE. If a third output file is needed, the 'A' is replaced by 'B'. This will continue for 'C', 'D', ... 'Z' for up to 27 output files (your original file plus files appended with A, B, ..., Z).

If your original output filespec had an eight-character filename, then the disassembler still closes the output file. However, you will have to enter a new filespec for the subsequent file. The following message will inform you of this action:

Output file is full; Enter filespec? >

If your new filespec contains an eight-character filename, you will continue to be prompted the next time another output file is needed. If, on the other hand, you enter a filespec with less than an eight-character name, automatic filespec generation will commence with the next output file, if required.

If your output disk becomes full and the disassembly is not complete, you will be prompted to change output diskettes - PROVIDED IT CAN BE ACCOMPLISHED. If the disk drive containing the output diskette is a hard drive [LDOS only], if you are running the disassembler from Job Control Language [LDOS only], or if the input is from a disk file located on the same disk drive as the output disk file, the disassembly will abort and the following message will be displayed:

Disk is full - Can't continue!

Otherwise, the disassembler will prompt you to change the output diskette with the following message:

Disk is full! - Enter new output disk <ENTER>

After replacing the output diskette, depress the <ENTER> key and the disassembler processing will continue.

DEVELOPING A SOURCE PROGRAM

The best way to employ the power of the MISOSYS disassembler in order to create a "SOURCE" program, the following steps should be performed:

1. Either disassemble directly from a disk CMD-type file (which will provide output of whatever is in the file) and proceed to step 2 or determine the boundaries of the machine language program in memory. If your target program is on a cassette tape, transfer it to a CMD disk file via a tape-to-disk utility then proceed to disassemble the disk file. If you insist on reading the tape program into memory by using the "S" control command, you will find that a better procedure is to use the "T" control command first since it is possible that the target program may load into the same region as the disassembler. The START and END values will automatically be initialized to those determined from the program tape itself.
2. Disassemble to the video screen or printer to detect regions that may have been character string literals or data. Make note of these regions on scratch paper (or the printer output listing) to use in building a screening data text file.
3. Follow up with a disk output command to generate the SOURCE disk file. Enter your screening data text filespec at the appropriate time so that the data regions of the target program are interpreted properly.
4. Load the SOURCE disk file into the editor assembler. If the output has been partitioned into multiple files, you will need to use the *GET facility of EDAS (or *INCLUDE facility of other assemblers).
5. Make an attempt to scrutinize the listing and comment those sections you begin to understand. As you become more experienced with Z-80 assembler code, this will become an easy task. Illogical code sequences are probably data areas that you omitted from the screening data text file.

DEVELOPING SCREENING DATA

The experienced assembly language programmer should have little difficulty in detecting the segments of a target program that are not code segments. However, since everyone cannot be considered "experienced", herewith are a few tips to keep in mind when you are screening a target program for non-code regions.

The first example illustrates an easy observation. Note that from the ASCII column, a message is spelled out starting from the label "M010E". A good guess is that the message extends through address X'011B' even though the last byte is a NOP instruction. Since the next label is not until after the NOP, you should first try to interpret from X'010E' through X'011B' as a literal. This is done by entering a screening data field as '\$10e-11b".

010B 5A	00145	M010B	LD	E,D	Z
010C 45	00146		LD	B,L	E
010D 00	00147		NOP		.
010E 52	00148	M010E	LD	D,D	R
010F 2F	00149		CPL		/
0110 53	00150		LD	D,E	S
0111 204C	00151		JR	NZ,M015F	L
0113 322042	00152		LD	(M4220),A	2 B
0116 41	00153		LD	B,C	A
0117 53	00154		LD	D,E	S
0118 49	00155		LO	C,C	I
0119 43	00156		LD	B,E	C
011A 0D	00157		DEC	C	.
011B 00	00158		NOP		.
011C Cs	00159	M011C	PUSH	BC	E

The next example illustrates nonsense code beginning at address X'07F8'. It appears to be nonsense through X'0808' but starting at X'0809' (which has a label), the code starts to make sense. This must be a data segment since there is no way to conceptualize literal data. The next question is whether it is "byte" or "word" data. Since there appears to be boundaries every four bytes (note the label positions), we may have byte, word, or even something else. Experience will show that the segment is actually floating-point data. However, you may treat it as "byte" data since "word" data will generate labels that may be spurious.

07F4 1F	01196		RRA		.
07F5 47	01197		LD	B,A	G
07F6 18EF	01198		JR	M07E7	.0
07F8 00	01199	M07F8	NOP		.
07F9 00	01200		NOP		.
07FA 00	01201		NOP		.
07FB 81	01202		ADD	A,C	.
07FC 03	01203	M07FC	INC	BC	.
07FD AA	01204		XOR	D	*
07FE 56	01205		LD	D,(HL)	V
07FF 19	01206		ADD	HL,DE	.
0800 80	01207	M0800	ADD	A,B	.
0801 F1	01208		POP	AF	q
0802 227680	01209		LD	(M8076),HL	"v."

0805 45	01210	LD	B,L	E
0806 AA	01211	XOR	D	*
0807 3882	01212	JR	C,M078B	8.
0809 CD5509	01213 M0809	CALL	M0955	MU.
080C B7	01214	OR	A	7
080D EA4A1E	01215	JP	PE,M1E4A	jJ.
0810 212441	01216	LD	HL,M4124	!\$A

The next example is something that you should quickly get familiar with. Note the interstitial label at address X'13E2'. It is pointing to an address that contains the byte, X'21'. If this were the beginning of an instruction, it would be a "LD HL,nn" 3-byte instruction. Also, if such were the case, the two following bytes would not be single-byte instructions but the operand field of the "LD HL,nn" instruction. You should also note that starting at X'13D8', the code is nonsense! You should easily observe that the segment from X'13D8' through X'13E1' is a data table. You should also quickly get used to those values - a power-of-10 table composed of the values, 10000, 1000, 100, 10, and 1. Set aside a data screening entry of "#13d8-13e1". This will also correct the decomposition of the "LD HL,nn" instruction.

13D2 A0	03053 M13D2	AND	B	.
13D3 86	03054	ADD	A,(HL)	.
13D4 011027	03055	LD	BC,M2710	..'
13D7 00	03056	NOP		.
13D8 1027	03057 M13D8	DJNZ	M1401	.'
13DA E8	03058	RET	PE	h
13DB 03	03059	INC	BC	.
13DC 64	06060	LD	H,H	d
13DD 00	03061	NOP		.
13DE 0A	03062	LD	A,(BC)	.
13DF 00	03063	NOP		.
13E0 010021	03064	LD	BC,M2100	..!
	03065 M13E2	EQU	\$-1	.
13E3 82	03066	ADD	A,D	.
13E4 09	03067	ADD	HL,BC	.

A great many tables of data relate to addresses. These are tables used as pointers or relocation data. They are easily picked out by nonsense code and logical values as "words". In the following partial listing, note that the first byte varies the full range of byte values (like a low-order value of an address) while the second byte is in a finite range (like the high-order byte of an address). Look for the scope of the table and you will have another "word" segment to be screened with a "#1608-bbbb" field.

1608 8A	03376 M1608	ADC	A,D	.
1609 09	03377	ADD	HL,BC	.
160A 37	03378	SCF		7
160B 0B	03379	DEC	BC	.
160C 77	03380	LD	(HL),A	w
160D 09	03381	ADD	HL,BC	.
160E D427EF	03382	CALL	NC,MEF27	T'o
1611 2AF527	03383	LD	HL,(M27F5)	*u'
1614 E7	03384	RST	20H	g

1615 13	03385	INC DE	.
1616 C9	03386	RET	I
1617 14	03387	INC D	.
1618 09	03388	ADD HL,BC	.
1619 08	03389	EX AF,AF'	.
161A 39	03390	ADD HL,SP	9

Programmers will, at times, provide "hidden" entry points by prefixing a one, two, or three byte instruction with a byte value that turns the instruction into a two, three, or four byte instruction. Its function is to mask the "hidden" instruction when reached in-line. Note the interstitial labels M199A, M199D, and M19A0. Doesn't the code look funny? Observe that the code is similar to the code at X'1997' if entered at the label entries. The code would, in fact, be additional "LD E,n" instructions with "n" taking on different values. Set up screening data fields as "1999,199c,199f" and the code will be properly decomposed. Incidentally, the label "M1998" was probably spuriously generated by decomposing a data region as code.

1991 2ADA40	04054	M1991 LD HL,(M40DA)	*Z@
1994 22A240	04055	LD (M40A2),HL	""@
1997 1E02	04056	M1997 LD E,02H	..
	04057	M1998 EQU \$-1	
1999 011E14	04058	LD BC,M141E	...
	04059	M199A EQU \$-2	
199C 011E00	04060	LD BC,M001E	...
	04061	M199D EQU \$-2	
199F 011E24	04062	LD BC,M241E	..\$
	04063	M19A0 EQU \$-2	
19A2 2AA240	04064	M19A2 LD HL,(M40A2)	*"@"
19A5 22EA40	04065	LD (M40EA),HL	"j@

The next example illustrates the same kind of "hidden" entry. Only this time, an extra byte of X'DD' is inserted to turn "LD HL,nn" instructions into "LD IX,nn" instructions. Register pair HL would be set to the value loaded at whatever entry was taken. Set up screening data as "24fa,24fe".

24F6 C9	00135	RET	I
24F7 21D225	00136	M24F7 LD HL,M25D2	!R%
24FA DD219825	00137	LD IX,M2598]!.%
	00138	M24FB EQU \$-3	
24FE DD217E25	00139	LD IX,M257E]!.%
	00140	M24FF EQU \$-3	
2502 3E0C	00141	LD A,0CH	>.
2504 EF	00142	RST 28H	0
2505 21FFFF	00143	LD HL,MFFFF	!..

These examples will put you on the right track when interpreting the disassembled output for data screening. Clean up literal segments first to reduce the frequency of spurious labels. Pay close attention to segments that do not make sense as code and screen as data. Analyze the reason for interstitial labels. Are they used to implement "hidden" entry points? In short time, you will become "experienced" in producing a near-perfect source document.

TEXT/BAS

=====

The following BASIC program may be used to enter screening text data into a file. It is a primitive program to be used only if you have no other text editor.

```

10 LINEINPUT "FILENAME/EXT"; F$: IF F$="" THEN CLOSE: END
20 OPEN "O",1,F$
30 PRINT"ENTER SCREENING DATA (NULL LINE TO END)"
40 LINEINPUT AS
50 IF A$ = "" THEN CLOSE: END
60 PRINT#1,A$: A$="": GOTO 40

```

MODEL I TRSDOS PATCH

=====

Model I TRSDOS users will find difficulty in reading the DSMBLR distribution disk (this is due to the data address mark used for the directory). Therefore, before making a BACKUP or copying files from the DSMBLR diskette, you will need to patch your TRSDOS 2.3. This change will in no way affect the operation of your TRSDOS. To prepare for this patch, obtain a fresh BACKUP of your TRSDOS 2.3. Enter the following BASIC program and RUN it.

```

10 OPEN"R",1,"SYS0/SYS.WKIA:0"
20 FIELD 1,171 AS R1$, 1 AS RS$, 84 AS R2$
30 GET 1,3: LSET RS$("<"): PUT 1,3: CLOSE: END

```

This program will change the byte in SYS0/SYS that loads at address X'46B0' from X'7C' to X'3C' . After you RUN the program, re-BOOT your TRSDOS diskette to correct the byte in memory.

Alternatively, if you do not want to patch your TRSDOS system diskette, you may change the value directly in memory by the use of DEBUG. This procedure is as follows:

1. At TRSDOS Ready, type DEBUG followed by <ENTER>.
2. Depress the <BREAK> key to enter the DEBUGger.
3. Type M46B0 followed by the <SPACE> bar.
4. Type 3C followed by <ENTER>.
5. Type G402D followed by <ENTER>.

Now, using either method noted above, COPY the files from the DSMBLR diskette to your TRSDOS diskette.